Certification-Monads*

Christian Sternagel and René Thiemann

April 19, 2020

Abstract

This entry provides several monads intended for the development of stand-alone certifiers via code generation from Isabelle/HOL. More specifically, there are three flavors of error monads (the sum type, for the case where all monadic functions are total; an instance of the former, the so called check monad, yielding either success without any further information or an error message; as well as a variant of the sum type that accommodates partial functions by providing an explicit bottom element) and a parser monad built on top. All of this monads are heavily used in the IsaFoR/CeTA project which thus provides many examples of their usage.

Contents

*This research is supported by FWF (Austrian Science Fund) projects J3202 and P22767.

1 Try-Catch and Error-Update Notation for Arbitrary Types

theory *Error-Syntax*
imports
Main
HOL−Library.Adhoc-Overloading
begin

consts
catch :: $'a \Rightarrow ('b \Rightarrow 'c) \Rightarrow 'c$ ((*try*(/ -)/ *catch*(/ -)) [*12*, *12*] *13*)
update-error :: $'a \Rightarrow ('b \Rightarrow 'c) \Rightarrow 'd$ (**infixl** <+*?* *61*)

syntax
-replace-error :: $'a \Rightarrow 'b \Rightarrow 'a$ (**infixl** <*?* *61*)

translations
m <*?* $e \rightharpoonup m$ <+*?* (λ-. e)

end

2 The Sum Type as Error Monad

theory *Error-Monad*
imports
HOL−Library.Monad-Syntax
Error-Syntax
begin

Make monad syntax (including do-notation) available for the sum type.

definition *bind* :: $'e + 'a \Rightarrow ('a \Rightarrow 'e + 'b) \Rightarrow 'e + 'b$
where
bind m f = (*case* m *of Inr* $x \Rightarrow f\ x$ | *Inl* $e \Rightarrow$ *Inl* e)

adhoc-overloading
Monad-Syntax.bind bind

abbreviation (*input*) *return* ≡ *Inr*
abbreviation (*input*) *error* ≡ *Inl*
abbreviation (*input*) *run* ≡ *projr*

2.1 Monad Laws

lemma *return-bind* [*simp*]:
(*return* x ⤜ f) = $f\ x$
⟨*proof*⟩

lemma *bind-return* [*simp*]:

$(m \gg\!= return) = m$
⟨*proof*⟩

lemma *error-bind* [*simp*]:
$(error\ e \gg\!= f) = error\ e$
⟨*proof*⟩

lemma *bind-assoc* [*simp*]:
fixes $m :: 'a + 'b$
shows $((m \gg\!= f) \gg\!= g) = (m \gg\!= (\lambda x.\ f\ x \gg\!= g))$
⟨*proof*⟩

lemma *bind-cong* [*fundef-cong*]:
fixes *m1 m2* $:: 'e + 'a$
and *f1 f2* $:: 'a \Rightarrow 'e + 'b$
assumes *m1* $=$ *m2*
and $\bigwedge y.\ m2 = Inr\ y \Longrightarrow f1\ y = f2\ y$
shows $(m1 \gg\!= f1) = (m2 \gg\!= f2)$
⟨*proof*⟩

definition *catch-error* $:: 'e + 'a \Rightarrow ('e \Rightarrow 'f + 'a) \Rightarrow 'f + 'a$
where
catch-def: $catch\text{-}error\ m\ f = (case\ m\ of\ Inl\ e \Rightarrow f\ e \mid Inr\ x \Rightarrow Inr\ x)$

adhoc-overloading
Error-Syntax.catch catch-error

lemma *catch-splits*:
$P\ (try\ m\ catch\ f) \longleftrightarrow (\forall e.\ m = Inl\ e \longrightarrow P\ (f\ e)) \wedge (\forall x.\ m = Inr\ x \longrightarrow P\ (Inr\ x))$
$P\ (try\ m\ catch\ f) \longleftrightarrow (\neg\ ((\exists e.\ m = Inl\ e \wedge \neg\ P\ (f\ e)) \vee (\exists x.\ m = Inr\ x \wedge \neg\ P\ (Inr\ x))))$
⟨*proof*⟩

abbreviation *update-error* $:: 'e + 'a \Rightarrow ('e \Rightarrow 'f) \Rightarrow 'f + 'a$
where
$update\text{-}error\ m\ f \equiv try\ m\ catch\ (\lambda x.\ error\ (f\ x))$

adhoc-overloading
Error-Syntax.update-error update-error

lemma *catch-return* [*simp*]:
$(try\ return\ x\ catch\ f) = return\ x$ ⟨*proof*⟩

lemma *catch-error* [*simp*]:
$(try\ error\ e\ catch\ f) = f\ e$ ⟨*proof*⟩

lemma *update-error-return* [*simp*]:
$(m <+?\ c = return\ x) \longleftrightarrow (m = return\ x)$

⟨*proof*⟩

definition *isOK m* ⟷ (*case m of Inl e* ⇒ *False* | *Inr x* ⇒ *True*)

lemma *isOK-E* [*elim*]:
assumes *isOK m*
obtains *x* **where** *m* = *return x*
⟨*proof*⟩

lemma *isOK-I* [*simp*, *intro*]:
m = *return x* ⟹ *isOK m*
⟨*proof*⟩

lemma *isOK-iff*:
isOK m ⟷ (∃ *x*. *m* = *return x*)
⟨*proof*⟩

lemma *isOK-error* [*simp*]:
isOK (*error x*) = *False*
⟨*proof*⟩

lemma *isOK-bind* [*simp*]:
isOK (*m* ⤜ *f*) ⟷ *isOK m* ∧ *isOK* (*f* (*run m*))
⟨*proof*⟩

lemma *isOK-update-error* [*simp*]:
isOK (*m* <+? *f*) ⟷ *isOK m*
⟨*proof*⟩

lemma *isOK-case-prod* [*simp*]:
isOK (*case lr of* (*l*, *r*) ⇒ *P l r*) = (*case lr of* (*l*, *r*) ⇒ *isOK* (*P l r*))
⟨*proof*⟩

lemma *isOK-case-option* [*simp*]:
isOK (*case x of None* ⇒ *P* | *Some v* ⇒ *Q v*) = (*case x of None* ⇒ *isOK P* | *Some v* ⇒ *isOK* (*Q v*))
⟨*proof*⟩

lemma *isOK-Let* [*simp*]:
isOK (*Let s f*) = *isOK* (*f s*)
⟨*proof*⟩

lemma *run-bind* [*simp*]:
isOK m ⟹ *run* (*m* ⤜ *f*) = *run* (*f* (*run m*))
⟨*proof*⟩

lemma *run-catch* [*simp*]:
isOK m ⟹ *run* (*try m catch f*) = *run m*
⟨*proof*⟩

fun *foldM* :: $('a \Rightarrow 'b \Rightarrow 'e + 'a) \Rightarrow 'a \Rightarrow 'b\ list \Rightarrow 'e + 'a$
where
foldM f d [] = *return d* |
foldM f d (*x* # *xs*) = *do* { *y* ← *f d x*; *foldM f y xs* }

fun *forallM-index-aux* :: $('a \Rightarrow nat \Rightarrow 'e + unit) \Rightarrow nat \Rightarrow 'a\ list \Rightarrow (('a \times nat) \times 'e) + unit$
where
forallM-index-aux P i [] = *return* () |
forallM-index-aux P i (*x* # *xs*) = *do* {
P x i <+? *Pair* (*x*, *i*);
forallM-index-aux P (*Suc i*) *xs*
}

lemma *isOK-forallM-index-aux* [*simp*]:
isOK (*forallM-index-aux P n xs*) = $(\forall i < length\ xs.\ isOK\ (P\ (xs\ !\ i)\ (i + n)))$
⟨*proof*⟩

definition *forallM-index* :: $('a \Rightarrow nat \Rightarrow 'e + unit) \Rightarrow 'a\ list \Rightarrow (('a \times nat) \times 'e) + unit$
where
forallM-index P xs = *forallM-index-aux P 0 xs*

lemma *isOK-forallM-index* [*simp*]:
isOK (*forallM-index P xs*) ⟷ $(\forall i < length\ xs.\ isOK\ (P\ (xs\ !\ i)\ i))$
⟨*proof*⟩

lemma *forallM-index* [*fundef-cong*]:
fixes *c* :: $'a \Rightarrow nat \Rightarrow 'e + unit$
assumes $\bigwedge x\ i.\ x \in set\ xs \Longrightarrow c\ x\ i = d\ x\ i$
shows *forallM-index c xs* = *forallM-index d xs*
⟨*proof*⟩

hide-const *forallM-index-aux*

Check whether *f* succeeds for all elements of a given list. In case it doesn't, return the first offending element together with the produced error.

fun *forallM* :: $('a \Rightarrow 'e + unit) \Rightarrow 'a\ list \Rightarrow ('a * 'e) + unit$
where
forallM f [] = *return* () |
forallM f (*x* # *xs*) = *f x* <+? *Pair x* ≫ *forallM f xs*

lemma *isOK-forallM* [*simp*]:
isOK (*forallM f xs*) ⟷ $(\forall x \in set\ xs.\ isOK\ (f\ x))$
⟨*proof*⟩

Check whether *f* succeeds for at least one element of a given list. In case it doesn't, return the list of produced errors.

fun *existsM* :: $('a \Rightarrow 'e + unit) \Rightarrow 'a\ list \Rightarrow 'e\ list + unit$
where
existsM f [] = *error* [] |
existsM f (*x* # *xs*) = (*try f x catch* ($\lambda e.$ *existsM f xs* <+? *Cons e*))

lemma *isOK-existsM* [*simp*]:
isOK (*existsM f xs*) $\longleftrightarrow$ ($\exists\, x{\in}set\ xs.$ *isOK* (*f x*))
⟨*proof*⟩

lemma *is-OK-if-return* [*simp*]:
isOK (*if b then return x else m*) $\longleftrightarrow$ $b \vee$ *isOK m*
isOK (*if b then m else return x*) $\longleftrightarrow$ $\neg\ b \vee$ *isOK m*
⟨*proof*⟩

lemma *isOK-if-error* [*simp*]:
isOK (*if b then error e else m*) $\longleftrightarrow$ $\neg\ b \wedge$ *isOK m*
isOK (*if b then m else error e*) $\longleftrightarrow$ $b \wedge$ *isOK m*
⟨*proof*⟩

lemma *isOK-if*:
isOK (*if b then x else y*) $\longleftrightarrow$ $b \wedge$ *isOK x* $\vee\ \neg\ b \wedge$ *isOK y*
⟨*proof*⟩

fun *sequence* :: $('e + 'a)\ list \Rightarrow 'e + 'a\ list$
where
sequence [] = *Inr* [] |
sequence (*m* # *ms*) = *do* {
x ← *m*;
xs ← *sequence ms*;
return (*x* # *xs*)
}

2.2 Monadic Map for Error Monad

fun *mapM* :: $('a \Rightarrow 'e + 'b) \Rightarrow 'a\ list \Rightarrow 'e + 'b\ list$
where
mapM f [] = *return* [] |
mapM f (*x*#*xs*) = *do* {
y ← *f x*;
ys ← *mapM f xs*;
Inr (*y* # *ys*)
}

lemma *mapM-error*:
($\exists\, e.$ *mapM f xs* = *error e*) $\longleftrightarrow$ ($\exists\, x{\in}set\ xs.\ \exists\, e.$ *f x* = *error e*)
⟨*proof*⟩

lemma *mapM-return*:
assumes *mapM f xs* = *return ys*

shows $ys = map\ (run \circ f)\ xs \wedge (\forall x{\in}set\ xs.\ \forall e.\ f\ x \neq error\ e)$
⟨*proof*⟩

lemma *mapM-return-idx*:
assumes $*$: $mapM\ f\ xs = Inr\ ys$ **and** $i < length\ xs$
shows $\exists y.\ f\ (xs\ !\ i) = Inr\ y \wedge ys\ !\ i = y$
⟨*proof*⟩

lemma *mapM-cong* [*fundef-cong*]:
assumes $xs = ys$ **and** $\bigwedge x.\ x \in set\ ys \Longrightarrow f\ x = g\ x$
shows $mapM\ f\ xs = mapM\ g\ ys$
⟨*proof*⟩

lemma *bindE* [*elim*]:
assumes $(p \gg\!= f) = return\ x$
obtains y **where** $p = return\ y$ **and** $f\ y = return\ x$
⟨*proof*⟩

lemma *then-return-eq* [*simp*]:
$(p \gg q) = return\ f \longleftrightarrow isOK\ p \wedge q = return\ f$
⟨*proof*⟩

fun *choice* :: $('e + 'a)\ list \Rightarrow 'e\ list + 'a$
where
$choice\ [] = error\ [] \mid$
$choice\ (x\ \#\ xs) = (try\ x\ catch\ (\lambda e.\ choice\ xs <+?\ Cons\ e))$

declare *choice.simps* [*simp del*]

lemma *isOK-mapM*:
assumes $isOK\ (mapM\ f\ xs)$
shows $(\forall x.\ x \in set\ xs \longrightarrow isOK\ (f\ x)) \wedge run\ (mapM\ f\ xs) = map\ (\lambda x.\ run\ (f\ x))\ xs$
⟨*proof*⟩

fun *firstM*
where
$firstM\ f\ [] = error\ []$
$\mid firstM\ f\ (x\ \#\ xs) = (try\ f\ x \gg return\ x\ catch\ (\lambda e.\ firstM\ f\ xs <+?\ Cons\ e))$

lemma *firstM*:
$isOK\ (firstM\ f\ xs) \longleftrightarrow (\exists x{\in}set\ xs.\ isOK\ (f\ x))$
⟨*proof*⟩

lemma *firstM-return*:
assumes $firstM\ f\ xs = return\ y$
shows $isOK\ (f\ y) \wedge y \in set\ xs$
⟨*proof*⟩

end

3 A Special Error Monad for Certification with Informative Error Messages

theory *Check-Monad*
imports *Error-Monad*
begin

A check is either successful or fails with some error.

type-synonym
$'e\ check = 'e + unit$

abbreviation *succeed* :: $'e\ check$
where
$succeed \equiv return\ ()$

definition *check* :: $bool \Rightarrow 'e \Rightarrow 'e\ check$
where
$check\ b\ e = (if\ b\ then\ succeed\ else\ error\ e)$

lemma *isOK-check* [*simp*]:
$isOK\ (check\ b\ e) = b\ \langle proof \rangle$

lemma *isOK-check-catch* [*simp*]:
$isOK\ (try\ check\ b\ e\ catch\ f) \longleftrightarrow b \vee isOK\ (f\ e)$
$\langle proof \rangle$

definition *check-return* :: $'a\ check \Rightarrow 'b \Rightarrow 'a + 'b$
where
$check\text{-}return\ chk\ res = (chk \gg return\ res)$

lemma *check-return* [*simp*]:
$check\text{-}return\ chk\ res = return\ res' \longleftrightarrow isOK\ chk \wedge res' = res$
$\langle proof \rangle$

lemma [*code-unfold*]:
$check\text{-}return\ chk\ res = (case\ chk\ of\ Inr\ \text{-} \Rightarrow Inr\ res \mid Inl\ e \Rightarrow Inl\ e)$
$\langle proof \rangle$

abbreviation *check-allm* :: $('a \Rightarrow 'e\ check) \Rightarrow 'a\ list \Rightarrow 'e\ check$
where
$check\text{-}allm\ f\ xs \equiv forallM\ f\ xs <+?\ snd$

abbreviation *check-exm* :: $('a \Rightarrow 'e\ check) \Rightarrow 'a\ list \Rightarrow ('e\ list \Rightarrow 'e) \Rightarrow 'e\ check$
where
$check\text{-}exm\ f\ xs\ fld \equiv existsM\ f\ xs <+?\ fld$

lemma *isOK-check-allm*:
isOK (*check-allm f xs*) ⟷ (∀ *x* ∈ *set xs*. *isOK* (*f x*))
⟨*proof*⟩

abbreviation *check-allm-index* :: ('*a* ⇒ *nat* ⇒ '*e check*) ⇒ '*a list* ⇒ '*e check*
where
check-allm-index f xs ≡ *forallM-index f xs* <+? *snd*

abbreviation *check-all* :: ('*a* ⇒ *bool*) ⇒ '*a list* ⇒ '*a check*
where
check-all f xs ≡ *check-allm* (λ*x*. *if f x then succeed else error x*) *xs*

abbreviation *check-all-index* :: ('*a* ⇒ *nat* ⇒ *bool*) ⇒ '*a list* ⇒ ('*a* × *nat*) *check*
where
check-all-index f xs ≡ *check-allm-index* (λ*x i*. *if f x i then succeed else error* (*x*, *i*)) *xs*

lemma *isOK-check-all-index* [*simp*]:
isOK (*check-all-index f xs*) ⟷ (∀ *i* < *length xs*. *f* (*xs* ! *i*) *i*)
⟨*proof*⟩

The following version allows to modify the index during the check.

definition
check-allm-gen-index ::
('*a* ⇒ *nat* ⇒ *nat*) ⇒ ('*a* ⇒ *nat* ⇒ '*e check*) ⇒ *nat* ⇒ '*a list* ⇒ '*e check*
where
check-allm-gen-index g f n xs = *snd* (*foldl* (λ(*i*, *m*) *x*. (*g x i*, *m* ≫ *f x i*)) (*n*, *succeed*) *xs*)

lemma *foldl-error*:
snd (*foldl* (λ(*i*, *m*) *x* . (*g x i*, *m* ≫ *f x i*)) (*n*, *error e*) *xs*) = *error e*
⟨*proof*⟩

lemma *isOK-check-allm-gen-index* [*simp*]:
assumes *isOK* (*check-allm-gen-index g f n xs*)
shows ∀ *x*∈*set xs*. ∃ *i*. *isOK* (*f x i*)
⟨*proof*⟩

lemma *check-allm-gen-index* [*fundef-cong*]:
fixes *f* :: '*a* ⇒ *nat* ⇒ '*e check*
assumes ⋀*x n*. *x* ∈ *set xs* ⟹ *g x n* = *g*′ *x n*
and ⋀*x n*. *x* ∈ *set xs* ⟹ *f x n* = *f*′ *x n*
shows *check-allm-gen-index g f n xs* = *check-allm-gen-index g*′ *f*′ *n xs*
⟨*proof*⟩

definition *check-subseteq* :: '*a list* ⇒ '*a list* ⇒ '*a check*
where
check-subseteq xs ys = *check-all* (λ*x*. *x* ∈ *set ys*) *xs*

lemma *isOK-check-subseteq* [*simp*]:
isOK (*check-subseteq xs ys*) ⟷ *set xs* ⊆ *set ys*
⟨*proof*⟩

definition *check-same-set* :: *'a list* ⇒ *'a list* ⇒ *'a check*
where
check-same-set xs ys = (*check-subseteq xs ys* ≫ *check-subseteq ys xs*)

lemma *isOK-check-same-set* [*simp*]:
isOK (*check-same-set xs ys*) ⟷ *set xs* = *set ys*
⟨*proof*⟩

definition *check-disjoint* :: *'a list* ⇒ *'a list* ⇒ *'a check*
where
check-disjoint xs ys = *check-all* (λ*x*. *x* ∉ *set ys*) *xs*

lemma *isOK-check-disjoint* [*simp*]:
isOK (*check-disjoint xs ys*) ⟷ *set xs* ∩ *set ys* = {}
⟨*proof*⟩

definition *check-all-combinations* :: (*'a* ⇒ *'a* ⇒ *'b check*) ⇒ *'a list* ⇒ *'b check*
where
check-all-combinations c xs = *check-allm* (λ*x*. *check-allm* (*c x*) *xs*) *xs*

lemma *isOK-check-all-combinations* [*simp*]:
isOK (*check-all-combinations c xs*) ⟷ (∀ *x* ∈ *set xs*. ∀ *y* ∈ *set xs*. *isOK* (*c x y*))
⟨*proof*⟩

fun *check-pairwise* :: (*'a* ⇒ *'a* ⇒ *'b check*) ⇒ *'a list* ⇒ *'b check*
where
check-pairwise c [] = *succeed* |
check-pairwise c (*x* # *xs*) = (*check-allm* (*c x*) *xs* ≫ *check-pairwise c xs*)

lemma *pairwise-aux*:
(∀ *j*<*length* (*x* # *xs*). ∀ *i*<*j*. *P* ((*x* # *xs*) ! *i*) ((*x* # *xs*) ! *j*))
= ((∀ *j*<*length xs*. *P x* (*xs* ! *j*)) ∧ (∀ *j*<*length xs*. ∀ *i*<*j*. *P* (*xs* ! *i*) (*xs* ! *j*)))
(**is** *?C* = (*?A* ∧ *?B*))
⟨*proof*⟩

lemma *isOK-check-pairwise* [*simp*]:
isOK (*check-pairwise c xs*) ⟷ (∀ *j*<*length xs*. ∀ *i*<*j*. *isOK* (*c* (*xs* ! *i*) (*xs* ! *j*)))
⟨*proof*⟩

abbreviation *check-exists* :: (*'a* ⇒ *bool*) ⇒ *'a list* ⇒ (*'a list*) *check*
where
check-exists f xs ≡ *check-exm* (λ*x*. *if f x then succeed else error* [*x*]) *xs concat*

lemma *isOK-choice* [*simp*]:
isOK (*choice* []) ⟷ *False*
isOK (*choice* (*x* # *xs*)) ⟷ *isOK x* ∨ *isOK* (*choice xs*)
⟨*proof*⟩

fun *or-ok* :: *'a check* ⇒ *'a check* ⇒ *'a check* **where**
or-ok (*Inl a*) *b* = *b* |
or-ok (*Inr a*) *b* = *Inr a*

lemma *or-is-or*: *isOK* (*or-ok a b*) = *isOK a* ∨ *isOK b* ⟨*proof*⟩

end

4 A Sum Type with Bottom Element

theory *Strict-Sum*
imports
HOL−Library.Monad-Syntax
Error-Syntax
Partial-Function-MR.Partial-Function-MR
begin

datatype (*dead 'e*, *'a*) *sum-bot* (**infixr** +⊥ *10*) = *Bottom* | *Left 'e* | *Right 'a* **for** *map*: *sum-bot-map*

4.1 Setup for Partial Functions

abbreviation *sum-bot-ord* :: *'e* +⊥ *'a* ⇒ *'e* +⊥ *'a* ⇒ *bool*
where
sum-bot-ord ≡ *flat-ord Bottom*

interpretation *sum-bot*:
partial-function-definitions sum-bot-ord flat-lub Bottom
⟨*proof*⟩

⟨*ML*⟩

4.2 Monad Setup

fun *bind* :: *'e* +⊥ *'a* ⇒ (*'a* ⇒ (*'e* +⊥ *'b*)) ⇒ *'e* +⊥ *'b*
where
bind Bottom f = *Bottom* |
bind (*Left e*) *f* = *Left e* |
bind (*Right x*) *f* = *f x*

lemma *bind-cong* [*fundef-cong*]:
assumes *xs* = *ys* **and** ⋀*x*. *ys* = *Right x* ⟹ *f x* = *g x*
shows *bind xs f* = *bind ys g*

⟨*proof*⟩

abbreviation *mono-sum-bot* :: $(('a \Rightarrow ('e +_\bot 'b)) \Rightarrow 'f +_\bot 'c) \Rightarrow bool$
where
mono-sum-bot ≡ *monotone* (*fun-ord sum-bot-ord*) *sum-bot-ord*

lemma *bind-mono* [*partial-function-mono*]:
assumes *mf*: *mono-sum-bot B* **and** *mg*: $\bigwedge y.$ *mono-sum-bot* $(\lambda f.\ C\ y\ f)$
shows *mono-sum-bot* $(\lambda f.\ bind\ (B\ f)\ (\lambda y.\ C\ y\ f))$
⟨*proof*⟩

adhoc-overloading
Monad-Syntax.bind bind

hide-const (**open**) *bind*

fun *catch-error* :: $'e +_\bot 'a \Rightarrow ('e \Rightarrow ('f +_\bot 'a)) \Rightarrow 'f +_\bot 'a$
where
catch-error Bottom f = *Bottom* |
catch-error (*Left a*) *f* = *f a* |
catch-error (*Right a*) *f* = *Right a*

adhoc-overloading
Error-Syntax.catch catch-error

lemma *catch-mono* [*partial-function-mono*]:
assumes *mf*: *mono-sum-bot B* **and** *mg*: $\bigwedge y.$ *mono-sum-bot* $(\lambda f.\ C\ y\ f)$
shows *mono-sum-bot* $(\lambda f.\ try\ (B\ f)\ catch\ (\lambda y.\ C\ y\ f))$
⟨*proof*⟩

definition *error* :: $'e \Rightarrow 'e +_\bot 'a$
where
[*simp*]: *error x* = *Left x*

definition *return* :: $'a \Rightarrow 'e +_\bot 'a$
where
[*simp*]: *return x* = *Right x*

fun *map-sum-bot* :: $('a \Rightarrow ('e +_\bot 'b)) \Rightarrow 'a\ list \Rightarrow 'e +_\bot 'b\ list$
where
map-sum-bot f [] = *return* [] |
map-sum-bot f (*x*#*xs*) = *do* {
y ← *f x*;
ys ← *map-sum-bot f xs*;
return (*y* # *ys*)
}

lemma *map-sum-bot-cong* [*fundef-cong*]:

assumes $xs = ys$ **and** $\bigwedge x.\ x \in set\ ys \Longrightarrow f\ x = g\ x$
shows *map-sum-bot f xs* = *map-sum-bot g ys*
⟨*proof*⟩

lemmas *sum-bot-const-mono* =
sum-bot.const-mono [*of fun-ord sum-bot-ord*]

lemma *map-sum-bot-mono* [*partial-function-mono*]:
fixes $C :: 'a \Rightarrow ('b \Rightarrow ('e +_\bot 'c)) \Rightarrow 'e +_\bot 'd$
assumes $\bigwedge y.\ y \in set\ B \Longrightarrow mono\text{-}sum\text{-}bot\ (C\ y)$
shows *mono-sum-bot* $(\lambda f.\ map\text{-}sum\text{-}bot\ (\lambda y.\ C\ y\ f)\ B)$
⟨*proof*⟩

abbreviation *update-error* $:: 'e +_\bot 'a \Rightarrow ('e \Rightarrow 'f) \Rightarrow 'f +_\bot 'a$
where
update-error r f ≡ *try r catch* $(\lambda\ e.\ error\ (f\ e))$

adhoc-overloading
Error-Syntax.update-error update-error

fun *sumbot* $:: 'e + 'a \Rightarrow 'e +_\bot 'a$
where
sumbot $(Inl\ x) = Left\ x$ |
sumbot $(Inr\ x) = Right\ x$

code-datatype *sumbot*

lemma [*code*]:
bind $(sumbot\ a)\ f = (case\ a\ of\ Inl\ b \Rightarrow sumbot\ (Inl\ b) \mid Inr\ a \Rightarrow f\ a)$
⟨*proof*⟩

lemma [*code*]:
$(try\ (sumbot\ a)\ catch\ f) = (case\ a\ of\ Inl\ b \Rightarrow f\ b \mid Inr\ a \Rightarrow sumbot\ (Inr\ a))$
⟨*proof*⟩

lemma [*code*]: *Right x* = *sumbot* $(Inr\ x)$ ⟨*proof*⟩

lemma [*code*]: *Left x* = *sumbot* $(Inl\ x)$ ⟨*proof*⟩

lemma [*code*]: *return x* = *sumbot* $(Inr\ x)$ ⟨*proof*⟩

lemma [*code*]: *error x* = *sumbot* $(Inl\ x)$ ⟨*proof*⟩

lemma [*code*]:
case-sum-bot f g h $(sumbot\ p)$ = *case-sum g h p*
⟨*proof*⟩

4.3 Connection to *Partial-Function-MR.Partial-Function-MR*

lemma *sum-bot-map-mono* [*partial-function-mono*]:
  assumes *mf*: *mono-sum-bot B*
  shows *mono-sum-bot* (λ*f*. *sum-bot-map h* (*B f*))
⟨*proof*⟩

⟨*ML*⟩

end

5 Monadic Parser Combinators

theory *Parser-Monad*
imports
  Error-Monad
  Show.Show
begin

abbreviation (*input*) *tab* ≡ *CHR 0x09*
abbreviation (*input*) *carriage-return* ≡ *CHR 0x0D*
abbreviation (*input*) *wspace* ≡ [*CHR* '' '', *CHR* ''↩'', *tab*, *carriage-return*]

definition *trim* :: *string* ⇒ *string*
  where *trim* = *dropWhile* (λ*c*. *c* ∈ *set wspace*)

lemma *trim*:
  ∃ *w*. *s* = *w* @ *trim s*
  ⟨*proof*⟩

A parser takes a list of tokes and returns either an error message or a result together with the remaining tokens.

type-synonym
  ('*t*, '*a*) *gen-parser* = '*t list* ⇒ *string* + ('*a* × '*t list*)

type-synonym
  '*a parser* = (*char*, '*a*) *gen-parser*

5.1 Monad-Setup for Parsers

definition *return* :: '*a* ⇒ ('*t*, '*a*) *gen-parser*
where
  return x = (λ*ts*. *Error-Monad.return* (*x*, *ts*))

definition *error* :: *string* ⇒ ('*t*, '*a*) *gen-parser*
where
  error e = (λ-. *Error-Monad.error e*)

definition *bind* :: ('t, 'a) *gen-parser* ⇒ ('a ⇒ ('t, 'b) *gen-parser*) ⇒ ('t, 'b) *gen-parser*
where
bind m f ts = *do* {
(*x*, *ts'*) ← *m ts*;
f x ts'
}

adhoc-overloading
Monad-Syntax.bind bind

lemma *bind-cong* [*fundef-cong*]:
fixes *m1* :: ('t, 'a) *gen-parser*
assumes *m1 ts2* = *m2 ts2*
and ⋀ *y ts*. *m2 ts2* = *Inr* (*y*, *ts*) ⟹ *f1 y ts* = *f2 y ts*
and *ts1* = *ts2*
shows ((*m1* ⤜ *f1*) *ts1*) = ((*m2* ⤜ *f2*) *ts2*)
⟨*proof*⟩

definition *update-tokens* :: ('t *list* ⇒ 't *list*) ⇒ ('t, 't *list*) *gen-parser*
where
update-tokens f ts = *Error-Monad.return* (*ts*, *f ts*)

definition *get-tokens* :: ('t, 't *list*) *gen-parser*
where
get-tokens = *update-tokens* (λ*x*. *x*)

definition *set-tokens* :: 't *list* ⇒ ('t, *unit*) *gen-parser*
where
[*code-unfold*]: *set-tokens ts* = *update-tokens* (λ-. *ts*) ≫ *return* ()

definition *err-expecting* :: *string* ⇒ ('t::*show*, 'a) *gen-parser*
where
err-expecting msg ts = *Error-Monad.error*
(''*expecting* '' @ *msg* @ ''*, but found*: '' @ *shows-quote* (*shows* (*take 30 ts*)) [])

fun *eoi* :: ('t :: *show*, *unit*) *gen-parser*
where
eoi [] = *Error-Monad.return* ((), []) |
eoi ts = *err-expecting* ''*end of input*'' *ts*

fun *exactly-aux* :: *string* ⇒ *string* ⇒ *string* ⇒ *string parser*
where
exactly-aux s i (*x* # *xs*) (*y* # *ys*) =
(*if x* = *y then exactly-aux s i xs ys*
else err-expecting (''''' @ *s* @ ''''') *i*) |
exactly-aux s i [] *xs* = *Error-Monad.return* (*s*, *trim xs*) |
exactly-aux s i (*x* # *xs*) [] = *err-expecting* (''''' @ *s* @ ''''') *i*

fun *oneof-aux* :: *string list* ⇒ *string list* ⇒ *string parser*
where
oneof-aux allowed (*x* # *xs*) *ts* =
(*if map snd* (*zip x ts*) = *x then Error-Monad.return* (*x*, *trim* (*List.drop* (*length x*) *ts*))
else oneof-aux allowed xs ts) |
oneof-aux allowed [] *ts* = *err-expecting* (''*one of* '' @ *shows-list allowed* []) *ts*

definition *is-parser* :: 'a *parser* ⇒ *bool* **where**
is-parser p ⟷ (∀ *s r x*. *p s* = *Inr* (*x*, *r*) ⟶ *length s* ≥ *length r*)

lemma *is-parserI* [*intro*]:
assumes ⋀*s r x*. *p s* = *Inr* (*x*, *r*) ⟹ *length s* ≥ *length r*
shows *is-parser p*
⟨*proof*⟩

lemma *is-parserE* [*elim*]:
assumes *is-parser p*
and (⋀*s r x*. *p s* = *Inr* (*x*, *r*) ⟹ *length s* ≥ *length r*) ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *is-parser-length*:
assumes *is-parser p* **and** *p s* = *Inr* (*x*, *r*)
shows *length s* ≥ *length r*
⟨*proof*⟩

A *consuming parser* (cparser for short) consumes at least one token of input.

definition *is-cparser* :: 'a *parser* ⇒ *bool*
where
is-cparser p ⟷ (∀ *s r x*. *p s* = *Inr* (*x*, *r*) ⟶ *length s* > *length r*)

lemma *is-cparserI* [*intro*]:
assumes ⋀*s r x*. *p s* = *Inr* (*x*, *r*) ⟹ *length s* > *length r*
shows *is-cparser p*
⟨*proof*⟩

lemma *is-cparserE* [*elim*]:
assumes *is-cparser p*
and (⋀*s r x*. *p s* = *Inr* (*x*, *r*) ⟹ *length s* > *length r*) ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *is-cparser-length*:
assumes *is-cparser p* **and** *p s* = *Inr* (*x*, *r*)
shows *length s* > *length r*
⟨*proof*⟩

lemma *is-parser-bind* [*intro*, *simp*]:

assumes *p*: *is-parser p* **and** *q*: $\bigwedge$ *x*. *is-parser* (*q x*)
shows *is-parser* (*p* $\gg\!=$ *q*)
⟨*proof*⟩

definition *oneof* :: *string list* ⇒ *string parser*
where
oneof xs = *oneof-aux xs xs*

lemma *oneof-result*:
assumes *oneof xs s* = *Inr* (*y*, *r*)
shows ∃ *w*. *s* = *y* @ *w* @ *r* ∧ *y* ∈ *set xs*
⟨*proof*⟩

definition *exactly* :: *string* ⇒ *string parser*
where
exactly s x = *exactly-aux s x s x*

lemma *exactly-result*:
assumes *exactly x s* = *Inr* (*y*, *r*)
shows ∃ *w*. *s* = *x* @ *w* @ *r* ∧ *y* = *x*
⟨*proof*⟩

hide-const *oneof-aux exactly-aux*

lemma *oneof-length*:
assumes *oneof xs s* = *Inr* (*y*, *r*)
shows *length s* ≥ *length y* + *length r* ∧ *y* ∈ *set xs*
⟨*proof*⟩

lemma *is-parser-oneof* [*intro*]:
is-parser (*oneof ts*)
⟨*proof*⟩

lemma *is-cparser-oneof* [*intro*, *simp*]:
assumes ∀ *x*∈*set ts*. *length x* ≥ *1*
shows *is-cparser* (*oneof ts*)
⟨*proof*⟩

lemma *exactly-length*:
assumes *exactly x s* = *Inr* (*y*, *r*)
shows *length s* ≥ *length x* + *length r*
⟨*proof*⟩

lemma *is-parser-exactly* [*intro*]:
is-parser (*exactly xs*)
⟨*proof*⟩

lemma *is-cparser-exactly* [*intro*]:
assumes *length xs* ≥ *1*

shows *is-cparser* (*exactly xs*)
⟨*proof*⟩

fun *many* :: (*char* ⇒ *bool*) ⇒ (*char list*) *parser*
where
many P (*t* # *ts*) =
(*if P t then do* {
(*rs*, *ts'*) ← *many P ts*;
Error-Monad.return (*t* # *rs*, *ts'*)
} *else Error-Monad.return* ([], *t* # *ts*)) |
many P [] = *Error-Monad.return* ([], [])

lemma *is-parser-many* [*intro*]:
is-parser (*many P*)
⟨*proof*⟩

definition *manyof* :: *char list* ⇒ (*char list*) *parser*
where
[*code-unfold*]: *manyof cs* = *many* (λ*c*. *c* ∈ *set cs*)

lemma *is-parser-manyof* [*intro*]:
is-parser (*manyof cs*)
⟨*proof*⟩

definition *spaces* :: *unit parser*
where
[*code-unfold*]: *spaces* = *manyof wspace* ≫ *return* ()

lemma *is-parser-return* [*intro*]:
is-parser (*return x*)
⟨*proof*⟩

lemma *is-parser-error* [*intro*]:
is-parser (*error x*)
⟨*proof*⟩

lemma *is-parser-If* [*intro*!]:
assumes *is-parser p* **and** *is-parser q*
shows *is-parser* (*if b then p else q*)
⟨*proof*⟩

lemma *is-parser-Let* [*intro*!]:
assumes *is-parser* (*f y*)
shows *is-parser* (*let x* = *y in f x*)
⟨*proof*⟩

lemma *is-parser-spaces* [*intro*]:
is-parser spaces
⟨*proof*⟩

fun *scan-upto* :: *string* ⇒ *string parser*
where
scan-upto end (*t* # *ts*) =
(*if map snd* (*zip end* (*t* # *ts*)) = *end then do* {
Error-Monad.return (*end*, *List.drop* (*length end*) (*t* # *ts*))
} *else do* {
(*res*, *ts'*) ← *scan-upto end ts*;
Error-Monad.return (*t* # *res*, *ts'*)
}) |
scan-upto end [] = *Error-Monad.error* (''*did not find end*−*marker* '' @ *shows-quote* (*shows end*) [])

lemma *scan-upto-length*:
assumes *scan-upto end s* = *Inr* (*y*, *r*)
shows *length s* ≥ *length end* + *length r*
⟨*proof*⟩

lemma *is-parser-scan-upto* [*intro*]:
is-parser (*scan-upto end*)
⟨*proof*⟩

lemma *is-cparser-scan-upto* [*intro*]:
is-cparser (*scan-upto* (*e* # *end*))
⟨*proof*⟩

end

6 More material on parsing

theory *Misc*
imports *Main*
begin

definition *span* :: ('*a* ⇒ *bool*) ⇒ '*a list* ⇒ '*a list* × '*a list*
where [*simp*]: *span P xs* = (*takeWhile P xs*, *dropWhile P xs*)

lemma *span-code* [*code*]:
span P [] = ([], [])
span P (*x* # *xs*) =
(*if P x then let* (*ys*, *zs*) = *span P xs in* (*x* # *ys*, *zs*) *else* ([], *x* # *xs*))
⟨*proof*⟩

definition *splitter* :: *char list* ⇒ *string* ⇒ *string* × *string*
where
[*code-unfold*]: *splitter cs s* = *span* (λ*c*. *c* ∈ *set cs*) *s*

end